# A Non-deterministic Deductive Database Language*

Yeh-Heng Sheng
UniSQL, Inc.
9380 Research Blvd.
Kaleido II, Ste. 220
Austin, TX 78759
(execu!sequoia!unisql!sheng@cs.utexas.edu)

## Abstract

A non-deterministic deductive database language IDLOG that employs tuple-identifiers in DATALOG (with negation) was proposed in [She90b] to enhance the expressive power of deductive database languages. It was shown that a subset of IDLOG programs defines the class of all computable deterministic queries. In this paper, we investigate the non-deterministic part of IDLOG. As discussed in [ASV90], the use of non-deterministic database languages is motivated using both pragmatic and theoretical considerations. There are natural non-deterministic queries whose implementation using deterministic languages is unintuitive and inefficient. One typical example is sampling queries, i.e., queries that randomly choose certain samples from a set of tuples, such as "Find an arbitrary set of employee samples that contains exactly N employees from each department (assuming each department has at least N employees)". Another consideration in favor of non-determinism is optimization. Intuitively, a non-deterministic program gives a certain degree of freedom in the computation of a query, which can be exploited in optimization. We show how IDLOG defines sampling queries and how it can be used to optimize DATALOG programs. Also discussed is the expressive power of non-deterministic IDLOG.

## 1 Introduction

A non-deterministic deductive database language IDLOG that employs tuple-identifiers in DATALOG (with negation) was proposed in [She90b] to enhance the expressive power of deductive database languages. It was shown that a subset of IDLOG programs defines the class of all computable deterministic queries. IDLOG is non-deterministic in the sense that we may get different answers for the same query, depending on the assignments of tuple-identifiers (tid's for short) in each relation. In this paper, we investigate the non-deterministic part of IDLOG.

As discussed in [ASV90], the use of non-deterministic database languages is motivated using both pragmatic and theoretical considerations. There are natural non-deterministic queries whose implementation using deterministic languages is unintuitive and inefficient. One typical example is sampling queries, i.e., queries that randomly choose certain samples from a set of tuples, such as "Find an arbitrary set of employee samples which contains exactly N employees from each department (assuming each department has at least N employees)". Any particular set that contains N students from each department will satisfy the first query. Deductive database languages, such as DATALOG, lack tools to define such queries. Recently, various non-deterministic database languages have been proposed for defining non-deterministic database queries[1] [ASV90, AV87, AV88, KN88, MS88]. Among these proposals, one important non-deterministic mechanism is the *choice operator* [KN88]. The choice operator is a very intuitive non-deterministic mechanism. The semantics of the choice operator is basically based on the minimal-model semantics, which gives it the advantage of making use of many existing evaluation strategies. However, the choice operator can not easily define sampling queries which ask for multiple samples (see Example 5). Sampling queries can be easily defined in our language. For example, the following IDLOG program defines the sampling query: "Find an arbitrary set of employee samples which contains exactly 2 employees from each department".

$select\_two\_emp(\text{Name}) \leftarrow emp[[2]](\text{Name}, \text{Dept}, \text{N}), \text{N} < 2$

*Most of the work was done at the State University of New York at Stony Brook



[1] Some of these languages can define database *updates*. Our attention in this paper, however, is on the query aspect of these languages.

where the literals *emp*[[2]](Name, Dept, N) and N < 2 specify that the two tuples that have tid 0 or 1 are to be considered in the computation of the query. We will show that IDLOG provides a basic framework for defining a more general notion of choice operators.

Another consideration in favor of non-determinism is optimization. Intuitively, a non-deterministic program yields a certain degree of freedom in the computation of a query, which can be exploited in optimization [ASV90]. For example, consider the following DATALOG program:

*all_depts*(Dept) ← *emp*(Name, Dept)

In order to compute *all_depts*, for each department $d$, only one tuple that contains $d$ in the second column needs to be considered. Argument positions, such as Name, were called existential in [RBK88]. It has been shown in [RBK88] that the detection of such existential arguments is in general undecidable. However, to optimize such programs, non-deterministic mechanisms can be used that provide users with explicit constructs for optimization. One of the purposes of the choice operator was to provide such an explicit construct. The previous DATALOG program can be recast into the following equivalent program with choice operator:

*all_depts*(Dept) ← *emp*(Name, Dept),
choice((Dept), (Name))

where the choice operator choice((Dept), (Name)) non-deterministically chooses one employee from each department. The DATALOG program can also be recast into the following equivalent IDLOG program:

*all_depts*(Dept) ← *emp*[[2]](Name, Dept, 0)

where the literal *emp*[[2]](Name, Dept, 0) specifies that only one particular employee that has tid 0 in each department will be considered.

In this paper, we define a new notion of existential arguments, called ∃-*existential* arguments, and provide an explicit language construct for specifying such arguments in programs. The number of intermediate redundant tuples in query evaluation can therefore be greatly reduced. The problem of identifying ∃-existential arguments in a DATALOG program is undecidable (Theorem 3). However, we note that the sufficient test for identifying existential arguments in [RBK88] works for ∃-existential arguments as well.

We also explore the expressive power of non-deterministic IDLOG. *Generic Turing machines*[2] introduced in [HS89, HS90] provide a convenient tool for us to show the expressive power of IDLOG. We show that the set of all non-deterministic queries defined by stratified IDLOG programs is equivalent to the set of all computable non-deterministic queries. The main contribution of this paper includes the following:

- We propose a non-deterministic deductive language that can easily define sampling queries[3]. The language generalizes DATALOG with choice operator.
- We introduce a new notion of existential arguments, and discuss its applications in optimizing DATALOG programs.
- We show that stratified IDLOG programs define all computable non-deterministic queries. This suggests that tuple identifiers may provide a powerful alternative in enhancing the expressive power of deductive database languages.

The rest of this paper is organized as follows. In Section 2, IDLOG is briefly described; Section 3 discusses non-deterministic database queries; The way IDLOG is used in optimizing DATALOG programs is shown in Section 4; Section 5 describes the expressive power of non-deterministic IDLOG; a concluding remark is given in Section 6. For brevity, we will omit all the proofs (they can be found in [She90a]).

# 2 Basics

In this section, we review the notion of *ID-relations*, and the syntax and semantics of IDLOG.

## 2.1 ID-relations

In this paper, we only consider *flat relations*, i.e. relations containing no complex objects such as sets and lists. Function symbols are excluded as well[4]. Consider a countably infinite set U, called the *universal domain*. An *uninterpreted domain* (u-domain for short) is a finite subset of U, and the *interpreted domain* is the set N of natural numbers 0, 1, 2, .... Suppose $m$, $n \geq 0$, and $D$ is a u-domain. The *types* defined in the following are 0, 1 sequences. A *relation* of *type* $s_1 \cdot \ldots \cdot s_m$ *over* $D$ is a (possibly infinite) subset of $D_1 \times \ldots \times D_m$ such that for $i = 1, \ldots, m$, $D_i = D$ if $s_i = 0$, and $D_i = \mathsf{N}$ if $s_i = 1$. Suppose for $j = 1, n$, $r_j$ is a finite relation of type $a_j$ over $D$. A *database* of type ($a_1$, ..., $a_n$) with u-domain $D$ is denoted by (u-domain=$D$; $r_1$, ..., $r_n$).

Intuitively, an ID-relation of a relation $r$ is obtained from $r$ by augmenting each tuple in $r$ with its tid that is determined according to the way tuples in $r$ are grouped. We define an *ID-function* of a relation $r$ as a one-to-one and onto function from $r$ to the set {0,

[2] Generic Turing machines were later renamed as *domain turing machines* in [HS90].

[3] As shown in [She90a], the language can also be efficiently implemented

[4] Although we are primarily interested in programs with no general function symbols, the IDLOG programs we define here can be easily extended to allow them.

$\ldots, |r| - 1\}$. A *sub-relation* of a relation $r$ grouped by a set $s$ of attributes of $r$ is a subset of $r$ that contains all the tuples in $r$ which have the same value on each attribute in $s$. Let $[t]_s$ be the sub-relation of $r$ grouped by $s$ that contains the tuple $t$. Now, suppose $r$ is an n-ary relation of type $a$, and $s$ is a subset of $\{1, \ldots, n\}$. An ID-relation of $r$ on $s$ is an $(n+1)$-ary relation of type $a \cdot 1$ over the same set as $r$, and its every tuple is composed of a tuple $t$ in $r$ and a tid which is assigned by an ID-function of the relation $[t]_s$.

**Example 1** Let $r = \{(a,c),(a,d),(b,c)\}$, then the sub-relations of $r$ grouped by the first attribute are:

$\{(a,c),(a,d)\}$, and

$\{(b,c)\}$

Therefore, there are two ID-relations of $r$ on $\{1\}$:

$\{(a,c,1),(a,d,0),(b,c,0)\}$, and

$\{(a,c,0),(a,d,1),(b,c,0)\}$. □

## 2.2 IDLOG: Syntax and Semantics

Consider a two-sorted first-order logic language [End72] with sorts u and i. Constants of sort u are elements of U and constants of sort i are 0, 1, .... Each *interpretation I* of this language contains two *universes*, namely u-universe and i-universe, where the latter is intended to be the set N. Equality symbols (for both sorts) are assumed in this language. We also assume the existence of a special predicate succ (or sort (i, i)) with the intended meaning: succ(A, B) iff B = A + 1. More complicated *arithmetic predicates*, such as +, −, ∗, / (of sort (i, i, i)), and < (of sort (i, i)), can be defined by IDLOG programs (defined later) using the predicate succ. Sorts of predicates are also written as 0, 1 sequences denoting attributes of sort u and sort i respectively. For convenience, we will not mention the sorts of variables and predicates if they can be inferred from the context. An *IDLOG language* $L$ is a such language with additional "ID-predicates". Let $s$ be a (possibly empty) subset of $\{1, \ldots, n\}$. For each ordinary $n$-ary predicate symbol $p$ of sort $a$, an *ID-version* of $p$ on $s$ is a predicate symbol of sort $a \cdot 1$, denoted by $p[\![s]\!]$[5]. The set of all *ID-predicates* in an IDLOG language $L$ is the set of all ID-versions of ordinary predicates in $L$. Hereinafter, unless stated otherwise, we will ignore arithmetic predicates, for their meaning is fixed.

*IDLOG interpretations* are extensions of first-order logic interpretations. Suppose $I$ is an IDLOG interpretation, and p is a predicate. The set $\mathsf{p}^I$ is the relation assigned by $I$ to p. An IDLOG interpretation $I$, in addition to the usual assignment of a relation to each ordinary predicate, assigns to each ID-predicate $p[\![s]\!]$ an ID-relation of $p^I$ on $s$. An atom (resp. a literal) is called an *ID-atom* (resp. an *ID-literal*) if it contains an ID-predicate, and a *non-ID-atom* (resp. a *non-ID-literal*) otherwise. An *IDLOG clause* is a universally quantified sentence of the form $(\forall^{s_1} X_1 \ldots \forall^{s_k} X_k)(A \leftarrow B_1 \wedge \ldots \wedge B_n)$, where the $B_j$'s are any literals, and $A$ is a non-ID-atom containing no succ or equality. An *IDLOG program* is a finite set of IDLOG clauses. Throughout this paper, we assume that each use of an arithmetic predicates is *safe* [Zan86] which is guaranteed through a sufficient condition: for each literal $l$ containing an arithmetic predicate in the body of a clause r, a "sufficient" number of arguments of $l$ must be *positively bound* in the body of the same clause, i.e., each of them either is a constant or appears in a positive literal containing no arithmetic predicates in the body of r. For example, in the following program,

$q$(a, 1)

$p1$(X, N) ← $q$(X, N), +(N, L, M)

$p2$(X, N) ← $q$(X, N), +(L, M, N)

the first occurrence of + is not allowed, since there are infinitely many solutions for the equation 1 + L = M, while the second occurrence of + is allowed since there are only finite number of solutions for the equation: L + M = 1. Thus, for the arithmetic predicate +, allowed combinations of bound (b) and unbound (n) arguments are "bbb", "bbn", "bnb", "nbb", and "nnb" (note that this is only a sufficient condition for ensuring safety of +). Similarly, we can define some sufficient numbers for all other arithmetic predicates.

Consider an IDLOG program $P$. The Herbrand universe of $P$, is the union of N and the set of all constants of sort u in $P$ (if there is no constant of sort u in $P$ then a new constant will be added). Henceforth, when mentioning formulae, we will assume each is of the right sort. Let $\mathsf{H}_P$ be the set of all predicates in $P$ and their corresponding ID-versions or ordinary predicates. The *Herbrand base* of $P$ consists of ground atoms constructed from all predicates in $\mathsf{H}_P$, and constants in the Herbrand universe of $P$. A *Herbrand interpretation* of $P$ is a subset of the Herbrand base of $P$ which satisfies the requirement of being an IDLOG interpretation, i.e. relations assigned to ordinary predicates and their ID-versions stand in right relationship. From now on, we will focus on Herbrand interpretations.

Recently, finding an appropriate declarative semantics for *logic programs* (DATALOG with negation and function symbols) has attracted much attention [ABW88, GL88, Prz88b, Prz88a, Gel88, VGRS88]. The following is the main result about perfect models of IDLOG programs.

**Theorem 1** *([She90b]) For every model N of a strati-*

[5] As pointed out by Richard Hull, among all the ID-predicates, those of the form $p[\![]$ (with no grouping attributes) are most primitive. That is, all other ID-predicates can be defined through them.

*fied IDLOG program $P$, there exists a perfect model $M$ of $P$ such that $M \preceq N$. Thus, every stratified IDLOG program has at least one perfect model.* □

The corresponding result in [Prz88b] for stratified logic programs can be thought of as a special case of the above theorem. The perfect-model semantics for IDLOG programs can therefore be similarly defined.

# 3 Non-deterministic Database Queries

In this section, we first define non-deterministic IDLOG queries, and give a brief overview of some non-deterministic database languages. Then we show how sampling queries can be defined in IDLOG.

## 3.1 Non-deterministic IDOLG Queries

The following definition of queries is an extension of the one defined in [CH80, Cha81]. Some terminology is borrowed from [HS89] and [AV88]. An *elementary* relation type is a relation type containing no 1's (i.e. all domains are uninterpreted). Suppose $O$ is a finite subset of U, and for $i \geq 0$, $a_i$ is an elementary relation type. A *(non-deterministic) query* $\mathbf{f}$ of type $\overline{a} = (a_1, \ldots, a_n) \rightarrow a_0/O$ is a binary relation between databases of type $\overline{a}$ and finite relations of type $a_0$ that satisfies the condition: if $(\mathbf{r}, r) \in \mathbf{f}$ then the relation must be over the union of $O$ and the u-domain of $\mathbf{r}$. For each $(\mathbf{r}, r) \in \mathbf{f}$, the database $\mathbf{r}$ is called an *input database* of $\mathbf{f}$, and the relation $r$ is called an *answer* of the query $\mathbf{f}$ on input $\mathbf{r}$ (or, for short, an answer of $\mathbf{f}(\mathbf{r})$). Note that since the set $O$ and each u-domain are finite, there are only finitely many possible answers for each particular input database. A query is *deterministic* if for each appropriate input, there is at most one answer. For convenience, we will also think of a query as a function from databases to sets of relations. Let $C$ be a finite subset of U. A query $\mathbf{f}$ of type $\overline{a} \rightarrow a_0/O$ is *$C$-generic* if it satisfies the following:
$r \in \mathbf{f}(\mathbf{r})$ iff $\sigma(r) \in \mathbf{f}(\sigma(\mathbf{r}))$ for every permutation $\sigma$ over U such that $\forall x \in C$, $\sigma(x) = x$ (here $\sigma$ is naturally extended to relations and databases).

Note that each $C$-generic query of type $\overline{a} \rightarrow a_0/O$ is also of type $\overline{a} \rightarrow a_0/C$. Thus, we will omit the type postfix when mentioning *generic queries* ($C$-generic queries for some $C$). A query $\mathbf{f}$ is *computable* iff it is *generic* and recursively enumerable. Note that when $\mathbf{f}$ is deterministic, $\mathbf{f}$ is computable iff it is generic and *Turing computable* (*partial recursive*).

Following [HS89], we briefly describe how *Turing machines* (*TM's*) are used to compute deterministic queries. Suppose $\mathbf{f}$ is a deterministic $C$-generic query of type $\overline{a} \rightarrow a_0$. A TM $M$ computes $\mathbf{f}$ if the following conditions are satisfied. First, it is assumed that $C$ and distinguished symbols '0', '1', ',', '(', ')', '[', and ']' are included in the tape-alphabet of $M$. An input database with u-domain $D$ is placed into an ordered list, where each uninterpreted constant in $D - C$ is encoded as a string of 0's and 1's. If $M$ halts, it must hold on its tape the encoding of a relation of type $a_0$ over $D \cup C$. Finally, the operation of $M$ must be independent of the encoding used for the uninterpreted constants in $D - C$, and also independent of the order in which the input is presented.

Suppose $\mathcal{P}$ is a stratified IDLOG program. An *input predicate* of $\mathcal{P}$ is any ordinary predicate $p$ such that $p$ does not appear in the head of a clause in $\mathcal{P}$, and $p$ or its ID-version occurs in the body of a clause; an *output predicate* of $\mathcal{P}$ is an ordinary predicate that appears in the head of a clause in $\mathcal{P}$. Note that the arithmetic predicates are not counted as either input or output predicates. Likewise, there is no need to pay special attention to arithmetic predicates when talking about finite models, since every use of computed predicates is assumed to be safe. A clause r is *related* to an output predicate $q$ of a program $\mathcal{P}$ if the head predicate of r appears either in a clause defining $q$ or, recursively, in a clause related to $q$. The *program portion* related to $q$ in $\mathcal{P}$, denoted $\mathcal{P}/q$, is the set of all clauses in $\mathcal{P}$ that are related to $q$. Now assume $q$ (of sort $a_0$) is an output predicate of $\mathcal{P}$, and $p_1$ (of sort $a_1$), ..., $p_n$ (of sort $a_n$) are input predicates of $\mathcal{P}$. Let $D = \{d_1, \ldots, d_m\}$ be a u-domain, and $\mathbf{r} = (\text{u-domain}=D; r_1, \ldots, r_n)$ be an input database for $\mathcal{P}$. The program
$\mathcal{P}/q \cup \{p_j(t) : 1 \leq j \leq n,\ t \in r_j,\ \&\ p_j \textit{ appears in } \mathcal{P}/q\}$
$\cup \{\mathsf{udom}(d_i) : i = 1, \ldots, m\}$
is called a *database program* of $\mathcal{P}$ w.r.t. the output predicate $q$ and the input database $\mathbf{r}$, written $\mathsf{dbp}(\mathcal{P}, q, \mathbf{r})$. Throughout this paper, we assume, in addition to the usual axioms of standard logic, the following axioms [Rei83] for the database program $\mathsf{dbp}(\mathcal{P}, q, \mathbf{r})$:

- *Equality Axioms*: the usual *reflexivity*, *commutativity*, and *transitivity* of equality symbols, plus the principle of substitution of equal terms,
- *Unique Name Axioms*: for each $i \neq j$, $d_i \neq d_j$,
- *Domain Closure Axiom*:
  $\forall^{\mathsf{u}} X\ (\vee \{X = a : a \in D$, or $a$ is an uninterpreted constant in $\mathcal{P}/q\})$.

Let $\mathsf{PERF}^f_{\mathcal{D}}$ be the set of all finite perfect (Herbrand) models of the database program $\mathcal{D} = \mathsf{dbp}(\mathcal{P}, q, \mathbf{r})$. Then $\mathcal{P}$ defines an IDLOG query $\mathbf{q}$ of type $(a_1, \ldots, a_n) \rightarrow a_0$ as follows:

$$\mathbf{q}(\mathbf{r}) = \begin{cases} \{q^I : I \in \mathsf{PERF}^f_{\mathcal{D}}\}, & \textit{if } \mathsf{PERF}^f_{\mathcal{D}} \neq \emptyset \\ \textit{undefined}, & \textit{otherwise} \end{cases}$$

Notice that query **q** is $C$-generic, where $C$ is the set of all uninterpreted constants in $\mathcal{P}$ (actually, the part of $\mathcal{P}$ that is related to $q$). We can similarly define queries defined by other deductive database languages. A program is *q-equivalent* to another program if they define the same query **q**.

**Example 2** Consider a clause

$man(X) \vee woman(X) \leftarrow person(X)$

and the queries **man** and **woman**. These queries can be defined in IDLOG as follows:

$sex\_guess(X, male) \leftarrow person(X)$

$sex\_guess(X, female) \leftarrow person(X)$

$man(X) \leftarrow sex\_guess[\![1]\!](X, male, 1)$

$woman(X) \leftarrow sex\_guess[\![1]\!](X, female, 1)$

The relation *sex_guess* contains every possible guess (*male* or *female*) of a person's sex; $sex\_guess[\![1]\!](X, male, 1)$ (resp., $sex\_guess[\![1]\!](X, female, 1)$) means that the guess $sex\_guess(X, male)$ (resp., $sex\_guess(X, male)$) is true (1). Suppose $r = \{(a), (b)\}$, and the input database is $\mathbf{r}$ = (u-domain=$\{a, b\}$; $r$). Now, the interpretation for the predicate *man* in each perfect model of the corresponding database program is one of the following: $\emptyset$, $\{man(a)\}$, $\{man(b)\}$, and $\{man(a), man(b)\}$. Therefore, the query **man** is evaluated to $\mathbf{man(r)} = \{\emptyset, \{(a)\}, \{(b)\}, \{(a), (b)\}\}$, so is the query **woman**. □

## 3.2 Overview

Database languages, such as first-order relational calculus and DATALOG, lack the ability to define queries that are non-deterministic in nature. Recently, various non-deterministic database languages have been proposed for defining database queries and updates [ASV90, AV87, AV88, MS88]. In the following discussion, we will focus on query aspects of these languages.

A fairly direct way to have a non-deterministic database language is to allow disjunctions in clause heads. As shown in [Prz88b], perfect-model semantics can be defined for such programs (denoted by DATALOG$^{\neg\vee}$). However, DATALOG$^{\neg\vee}$ does not provide a convenient mechanism for defining sampling queries, and it is unlikely that the language enhances the expressive power of DATALOG$^{\neg\vee}$ in defining deterministic queries. Another direct way of having a non-deterministic database language is to consider the class of all logic programs based on *stable models* [GL88, SZ90]. Because IDLOG actually defines all computable non-deterministic queries (shown in Theorem 6), every query defined by a non-stratified logic program based on stable model semantics can also be defined by a stratified IDLOG program [She90a]. In the rest of this section, we will only discuss two non-deterministic mechanisms, namely the non-deterministic inflationary semantics [AV88, ASV90] and the choice operator [KN88, NT89].

### 3.2.1 Non-deterministic Inflationary Semantics

We will discuss two closely related non-deterministic database languages, each of which has DATALOG-like syntax. The semantics of both languages is based on a *non-deterministic inflationary* semantics. This means that the intended models of programs are obtained by applying program clauses bottom up, each clause is instantiated one at a time, and facts are added to the output until no additional facts can be inferred (no facts are ever deleted). Notice that choice available in consecutive instantiations of rules introduces non-determinism, and therefore the output varies depending on this choice.

The language *DL* (a Declarative Language) introduced in [AV88] has DATALOG syntax extended by allowing (1) negative literals in the body, and (2) more than one positive literal in the head (connected by the logical AND). Variables appearing in the head but not in the body are interpreted as the source of *invented values*, i.e., instantiations of such values are new constants not appearing in the program or in the current computation of the semantics. The random assignment of new values to such variables is another cause of non-determinism. Queries defined by DL programs are understood w.r.t. the *inflationary fixpoints* [GS85, KP88] of these programs instead of perfect models as it was in the DATALOG$^{\neg}$ case.

An *N-DATALOG*$^{\neg*}$ program [ASV90] resembles a DL program except that negative literals are allowed in the head of each clause, and each variable occurring in the head of a clause has to appear positively bound in the body as well. Negations in heads of clauses are interpreted as *deletions*. An instantiation of a clause can only be fired if its head is consistent, i.e., it contains no literal together with its negation. Notice that allowing negations only in the head (but not in the body) does not introduce any non-determinism [ASV90].

**Example 3** Consider the following DL program:

$man(X) \leftarrow person(X), \neg\ woman(X)$

$woman(X) \leftarrow person(X), \neg\ man(X)$

Suppose the input database **r** contains $person(a)$ and $person(b)$. Then, under the non-deterministic inflationary semantics, $\mathbf{man(r)} = \mathbf{woman(r)} = \{\emptyset, \{(a)\}, \{(b)\}, \{(a), (b)\}\}$. In contrast, under the deterministic inflationary semantics, $\mathbf{man(r)} = \mathbf{woman(r)} = \{(a), (b)\}$. □

The non-deterministic inflationary semantics is important in exploring the expressive power of database languages. The object of this paper, however, is to

provide a general non-deterministic database language as a simple extension of DATALOG based on the minimal-model semantics and perfect-model semantics. By staying within this semantics paradigm, many existing evaluation strategies for logic programs can be used for evaluating IDLOG programs with only slight modifications.

### 3.2.2 The Choice Operator

The *choice operator* choice introduced in [KN88] (see also [NT89]) provides users with an explicit non-deterministic construct. In the following discussions, the terminology used slightly differs from [KN88]. Consider the following DATALOG program $\mathcal{P}$ with choice (or *DATALOG$^C$ program* for short) ([KN88]):

*select_emp*(Name) ← *emp*(Name, Dept), choice((Dept), (Name))

$\mathcal{P}$ computes the set of all employees that contains exactly one employee from each department. The choice operator choice((Dept), (Name)) non-deterministically chooses one employee from each department. That is, let $r$ be the relation that contains all tuples (Dept, Name) satisfying the body (without the choice operator) of the clause. Then the choice operator chooses a subset $r_1$ from $r$ such that $r_1$ contains all departments and satisfies the functional dependency "Dept ⇒ Name". Let's call such a set $r_1$ a *functional subset* of the relation $r$ w.r.t. the attributes Dept and Name. In defining the semantics of DATALOG$^C$ programs, each DATALOG$^C$ program $\mathcal{P}$ is first translated into a DATALOG program $\mathcal{P}^c$ by

1. replacing each occurrence of a choice operator choice($\overline{X}$, $\overline{Y}$) in some clause r ∈ $\mathcal{P}$ by a literal extChoice$_i$($\overline{X}$, $\overline{Y}$), where extChoice$_i$ is a new predicate symbol, called a *choice-predicate*, and

2. for each such occurrence, adding the following *choice-clause*:
   extChoice$_i$($\overline{X}$, $\overline{Y}$) ← *body*
   where *body* is the body of r without the choice operator.

To define the semantics of choice properly, we might impose the following syntactic restrictions on DATALOG$^C$ programs: in a DATALOG$^C$ program $\mathcal{P}$,

(C1) every clause contains at most one choice operator,

(C2) every clause containing a choice operator is not related to the head predicate of another clause that contains a choice operator. i.e., for any two clauses in $\mathcal{P}$ of the form:
[1] $p(\ldots)$ ← …, choice(…)
[2] $q(\ldots)$ ← …, choice(…)
the clause [1] $\notin \mathcal{P}/q$, and the clause [2] $\notin \mathcal{P}/p$.

Notice that C1 above does not really limit the expressive power of DATALOG$^C$. The reason why we need C2 is because the semantics defined in [KN88] does not seem to be appropriate for all DATALOG$^C$ programs (see [She90a]), but only for the subset of DATALOG$^C$ programs satisfying the above conditions. Then each *intended model* $N$ of $\mathcal{P}$ is constructed as follows:

- Construct the unique minimal model $M$ of $\mathcal{P}^c$.
- for each choice-predicate extChoice$_i$ in a clause *head* ← *body*, extChoice$_i$($\overline{X}$, $\overline{Y}$) in $\mathcal{P}^c$, assign to extChoice$_i$ a functional subset, say $s_i$, of the relation extChoice$_i^M$ w.r.t. $\overline{X} \cup \overline{Y}$.
- Construct the unique minimal model of the DATALOG program containing each non-choice-clause in $\mathcal{P}^c$ and every fact extChoice$_i(t)$, where $t \in s_i$.

The resulting set is an intended model of the DATALOG$^C$ program $\mathcal{P}$. For example, the following DATALOG$^C$ program is *man*-equivalent and *woman*-equivalent to the IDLOG program in Example 2.

*sex_guess*(X, *male*) ← *person*(X)
*sex_guess*(X, *female*) ← *person*(X)
*sex*(X, Y) ← *sex_guess*(X, Y), choice((X), (Y))
*man*(X) ← *sex*(X, *male*)
*woman*(X) ← *sex*(X, *female*)

Thus, IDLOG can be thought of as a general framework for implementing the choice operator in DATALOG:

**Theorem 2** *For every DATALOG$^C$ program $\mathcal{P}$ and every output predicate $q$ of $\mathcal{P}$, if $\mathcal{P}$ satisfies the conditions (C1) and (C2), then there exists a stratified four-stratum IDLOG program $\mathcal{Q}$ such that $\mathcal{P}$ and $\mathcal{Q}$ are $q$-equivalent.* □

## 3.3 Sampling Queries

In this section, we show how sampling queries can be defined in IDLOG. *Sampling queries* are (non-deterministic) queries that randomly choose samples from a set of tuples, such as "Find an arbitrary set of employee samples that contains exactly N employees from each department (assuming each department has at least N employees)", and "Find an arbitrary cafe at the intersection of Blvd. St. Germain and Blvd. St. Michel [ASV90]". When the number of samples to be selected is exactly one, sampling queries can be naturally defined by the choice operator:

**Example 4** Consider the sampling query "Find an arbitrary set of employee samples that contains exactly one employee from each department" [KN88]. It can be defined by the DATALOG$^C$ program:

*select_emp*(Name) ← *emp*(Name, Dept),
choice((Dept), (Name))
It can also defined by the following IDLOG program:
*select_emp*(Name) ← *emp*[[2]](Name, Dept, 0)
□

However, general sampling queries usually ask for multiple samples, which cannot be easily defined by DATALOG$^C$ programs:

**Example 5** Consider the sampling query "Find an arbitrary set of employee samples that contain exactly two employees from each department (assuming each department has at least two employees)". Notice that the following DATALOG$^C$ program does not really define this query:
*emp*1(Name, Dept) ← *emp*(Name, Dept),
choice((Dept), (Name))
*emp*2(Name, Dept) ← *emp*(Name, Dept),
choice((Dept), (Name))
*select_two_emp*(Name1) ← *emp*1(Name1, Dept),
*emp*2(Name2, Dept), Name1 ≠ Name2
In the above program, every occurrence of choice is independent from the other. There are some intended models of this program that contains exactly two students from each department, while others may not contain any student from a certain department. Although the purpose can be achieved by a modified version of this program in a two-sorted logic program as shown in [She90a], a considerable amount of overhead in defining this sampling query may not be avoidable. For example, for samples of size $n$, we have to choose one employee from each department $n$ times, and then perform $n(n-1)/2$ tests of inequalities. The same query can be more easily defined by the IDLOG program:
*select_two_emp*(Name) ← *emp*[[2]](Name, Dept, N), N < 2
where each tuple with tid 0 or tid 1 in each sub-relation of *emp* grouped by Dept is chosen.[6] □

The inadequacy of defining general sampling queries by the choice operator motivates the need of having *multiple-choice operators*, such as choice2 choosing two samples, and choice3 choosing three samples. Yet, each such multiple-choice operator requires a different interpretation based on the DATALOG$^C$ semantics. However, IDLOG can be thought of as a natural framework for expressing these operators.

[6] Indeed, as shown in [She90a], the condition "N < 2" can be used to generate an optimization information which ensures that only two tuples of the relation *emp* will be used in the evaluation.

# 4 Optimizing DATALOG Programs

One important issue in optimizing deductive database programs is eliminating redundant solutions. For example, consider the following DATALOG program [KN88] $\mathcal{P}$:
$p$(X) ← $q$(X, Z), $x$(Z, Y), $y$(W)
In order to compute $p$, we need not compute $y$ beyond determining whether $y$ is non-empty. Moreover, for a given tuple (X, Z) in $q$, only one tuple in $x$ containing Z in the first column needs to be considered. Argument positions, such as Y and W were called *existential* in [RBK88].

**Definition 1** Suppose $\mathcal{P}$ is a DATALOG program containing no facts, and $q$ is an output predicate of $\mathcal{P}$. Suppose also $p(\overline{X}, Y)$ is a literal in the body of a clause r in $\mathcal{P}$. Then the argument position corresponding to $Y$ in $p(\overline{X}, Y)$ is called an *existential argument w.r.t.* $q$ if the following holds:

> Let a new clause $p'(\overline{X}, Y') \leftarrow p(\overline{X}, Y)$ be added to $\mathcal{P}$, and let $p(\overline{X}, Y)$ be replaced by $p'(\overline{X}, Y')$ in r. Further, let any occurrence of $Y$ in the head predicate of r be replaced by $Y'$. Then, the new program is $q$-equivalent to $\mathcal{P}$.

Suppose $q$ is the head predicate of r. Then an argument of $q$ is existential if the argument is existential in all occurrences of $q$ in the bodies of clauses in $\mathcal{P}$. □

It has been shown in [RBK88] that the detection of existential arguments is in general undecidable. The adornment algorithm (a sufficient test) provided in [RBK88] for identifying existential arguments is based on the property that if a variable $Y$ appears in a body literal and does not appear anywhere else in the clause, except possibly in an existential argument of the head, then the argument position corresponding to $Y$ is existential. Except the arguments in input predicates, each argument identified as existential by this algorithm can be eliminated without affecting the query defined by the underlying output predicate, which is basically an operation of "pushing projections".

**Example 6** [RBK88]
Consider the following program:
[1] $q$(X) ← $a$(X, Y)
[2] $a$(X, Y) ← $p$(X, Z), $a$(Z, Y)
[3] $a$(X, Y) ← $p$(X, Y)
According to the algorithm, the second argument $Y$ of $a$ in [1] will be identified as existential because $Y$ does not appear anywhere else, which implies that the occurrences of $Y$ in the heads of [2] and [3] are existential. Now the second argument $Y$ of $a$ in the body

of [2] is also identified as existential since it does not appear anywhere else except in the existential argument of the head. Similarly, the second argument of $p$ in [3] is existential. The program can be recast into the following $q$-equivalent program:

$q(\mathrm{X}) \leftarrow a'(\mathrm{X})$
$a'(\mathrm{X}) \leftarrow p(\mathrm{X}, \mathrm{Z}), a'(\mathrm{Z})$
$a'(\mathrm{X}) \leftarrow p(\mathrm{X}, \mathrm{Y})$

□

In the following, we introduce another definition of existential arguments. Intuitively, the difference between the new definition and the previous one is the new relation used for replacing the relation containing existential arguments. Under the new definition, the relation ($p[\![s]\!](\overline{X}, Y, 0)$ in the following definition) consists of "exactly one tuple" (from every sub-relation of the relation to be replaced), while under the previous definition, the relation ($p'(\overline{X}, Y')$ in Definition 1) contains "all tuples".

**Definition 2** Suppose $\mathcal{P}$ is a DATALOG program containing no facts, and $q$ is an output predicate of $\mathcal{P}$. Suppose also $p(\overline{X}, Y)$ is a literal in the body of a clause r in $\mathcal{P}$, and $s$ contains the argument positions corresponding to $\overline{X}$. Then the argument position corresponding to $Y$ in $p(\overline{X}, Y)$ is called an *existential argument w.r.t.* $q$ if $\mathcal{P}$ is $q$-equivalent to the IDLOG program obtained from $\mathcal{P}$ by replacing the literal $p(\overline{X}, Y)$ in r by the ID-literal $p[\![s]\!](\overline{X}, Y, 0)$. □

For ease of discussion, existential arguments defined by Definition 1 will be mentioned as *∀-existential arguments*, while the ones defined by Definition 2 are called *∃-existential arguments*. As shown in the following example, the two notions are different.

**Example 7** Consider the program $\mathcal{P}$ given by
[1] $q_1 \leftarrow x(\mathrm{c})$
[2] $q_2 \leftarrow x(\mathrm{a})$
[3] $x(\mathrm{Y}) \leftarrow p(\mathrm{Y})$
[4] $p(\mathrm{b}) \leftarrow y(\mathrm{X})$
[5] $p(\mathrm{c}) \leftarrow y(\mathrm{X})$
The query $\mathbf{q_1}$ defined by $\mathcal{P}$ returns TRUE if and only if the input for predicate $y$ is not empty; the query $\mathbf{q_2}$ defined by $\mathcal{P}$ always returns FALSE. We will show that the argument position corresponding to Y in the body of clause [3] is ∀-existential but not ∃-existential w.r.t. $q_1$, and ∃-existential but not ∀-existential w.r.t. $q_2$.

1. The argument is ∀-existential w.r.t. $q_1$ since $\mathcal{P}$ is $q_1$-equivalent to the program $\mathcal{P}_1$:
   $q_1 \leftarrow x(\mathrm{c})$
   $q_2 \leftarrow x(\mathrm{a})$
   $x(\mathrm{Y}') \leftarrow p'(\mathrm{Y}')$
   $p'(\mathrm{Y}') \leftarrow p(\mathrm{Y})$
   $p(\mathrm{b}) \leftarrow y(\mathrm{X})$
   $p(\mathrm{c}) \leftarrow y(\mathrm{X})$
   But, since the query $\mathbf{q_2}$ defined by $\mathcal{P}_1$ returns TRUE on non-empty inputs, $\mathcal{P}$ is not $q_2$-equivalent to $\mathcal{P}_1$. Therefore, the argument is not ∀-existential w.r.t. $q_2$.

2. Let the following IDLOG program be $\mathcal{P}_2$.
   $q_1 \leftarrow x(\mathrm{c})$
   $q_2 \leftarrow x(\mathrm{a})$
   $x(\mathrm{Y}) \leftarrow p[\![]\!](\mathrm{Y}, 0)$
   $p(\mathrm{b}) \leftarrow y(\mathrm{X})$
   $p(\mathrm{c}) \leftarrow y(\mathrm{X})$
   Depending on which tuple ((b) or (c)) is assigned tid 0 in $p[\![]\!]$, the query $\mathbf{q_1}$ defined by $\mathcal{P}_2$ may return TRUE or FALSE on non-empty inputs (i.e., the query $\mathbf{q_1}$ is not deterministic). Therefore, it is not ∃-existential w.r.t. $q_1$. However, the query $\mathbf{q_2}$ defined by $\mathcal{P}_2$ always returns FALSE no matter what the input is. The argument is hence ∃-existential w.r.t. $q_2$.

□

The definition of ∃-existential arguments suggests the following optimization strategy for DATALOG programs: every literal containing an ∃-existential argument can be replaced by an ID-literal mentioned in Definition 2. This may greatly reduce the number of redundant tuples. Unfortunately, the problem of detecting ∃-existential arguments is in general undecidable as well.

**Theorem 3** *The problem of identifying ∃-existential arguments in DATALOG programs is undecidable.* □

Despite the dissimilarity between ∃- and ∀-existential arguments, there is a correlation between them:

**Theorem 4** *Every ∀-existential argument identified by the adornment algorithm in [RBK88] is also an ∃-existential argument.* □

For example, the program $\mathcal{P}$ at the beginning of this section can be recast into the following $p$-equivalent IDLOG program:
$p(\mathrm{X}) \leftarrow q(\mathrm{X}, \mathrm{Z}), x[\![1]\!](\mathrm{Z}, \mathrm{Y}, 0), y[\![]\!](\mathrm{W}, 0)$
This suggests the following optimization strategy, where the first two steps are taken from [RBK88]:

1. Use the adornment algorithm in [RBK88] to identify existential arguments, and transform the program accordingly (w.r.t. a certain output predicate).

2. Eliminate each identified existential argument appearing in an output predicate.

3. Suppose $p$ is an input predicate, $p(\overline{Y})$ is a literal in the program, and $X_1, \ldots, X_n$ are existential arguments in $p(\overline{Y})$. Then replace $p(\overline{Y})$ by the ID-literal $p[\![s]\!](\overline{Y}, 0)$, where $s$ corresponds to arguments in $\overline{Y} - \{X_1, \ldots, X_n\}$.

4. The resulting program can then be optimized by Algorithm D.1 in [She90a].[7]

**Example 8** The program in Example 6 can be further recast into the following $q$-equivalent program $\mathcal{P}$:

$q(\mathrm{X}) \leftarrow a'(\mathrm{X})$
$a'(\mathrm{X}) \leftarrow p(\mathrm{X}, \mathrm{Z}), a'(\mathrm{Z})$
$a'(\mathrm{X}) \leftarrow p[\![1]\!](\mathrm{X}, \mathrm{Y}, 0)$

Note that the second clause of this program can actually be discarded without affecting the query **q** defined by this program as shown in [RBK88]. But this is beyond the scope of this paper. □

With this optimization strategy, ID-predicates can be used to optimize DATALOG programs to a certain degree. Yet, more importantly, they provide users with an explicit construct for specifying ∃-existential arguments which cannot be effectively detected. One of the motivations for introducing the choice operator to DATALOG is also to provide users with an explicit construct for removing redundant solutions. Our results in the previous section suggest that the choice operators are really designated for ∃-existential arguments but not ∀-existential arguments. The relationship between choice and cut in top-down evaluation was also discussed in [KN88]. It is known that every DATALOG program with cut has an equivalent DATALOG$^C$ program. Since IDLOG subsumes DATALOG$^C$, it means that cut can be expressed in IDLOG as well.

# 5 The Expressive Power of Non-deterministic IDLOG

Due to limited space, in this section, we just list some results about the expressive power of IDLOG in expressing non-deterministic queries. We make use *generic Turing machines* [HS89] to show that the class of queries defined by stratified IDLOG programs is equivalent to the class of all computable non-deterministic queries.

**Theorem 5** *([HS89]) The class of queries computed by input-order independent generic Turing machines is equivalent to the class of computable deterministic queries.* □

[7] The tid 0 in the previous step can be used to generate an optimization information which can be used to ensure that only one tuple of the relation $p$ will be used in the evaluation.

**Lemma 1** *The class of non-deterministic queries computed by input-order independent NGTM's is equivalent to the class of computable non-deterministic queries.*

**Theorem 6** *The class of non-deterministic queries defined by stratified IDLOG programs is equivalent to the class of computable non-deterministic queries.* □

# 6 Conclusion

In this paper, we studied the non-deterministic part of IDLOG. Our language generalizes DATALOG with choice operator in terms of defining sampling queries and optimizing DATALOG programs. A new notion of ∃-existential arguments is introduced. IDLOG provides an explicit construct for specifying such arguments in programs such that the number of intermediate redundant tuples in query evaluation can be greatly reduced. We showed that the problem of identifying ∃-existential arguments in a DATALOG program is undecidable, but noted that the sufficient test for identifying ∀-existential arguments given in [RBK88] works for ∃-existential arguments as well. We have also shown that stratified IDLOG programs define all computable non-deterministic queries.

# Acknowledgments

The author is greatly indebted to Michael Kifer for valuable discussions and suggestions which considerably improved this paper. Thanks are also due to David Scott Warren for his helpful comments.

# References

[ABW88] K.R. Apt, H.A. Blair, and A. Walker. Towards a Theory of Declarative Knowledge. *Foundations of Deductive Database and Logic Programming (ed.)*, 1988.

[ASV90] S. Abiteboul, E. Simon, and V. Vianu. Non-Deterministic Languages to Express Deterministic Transformation. *Proceedings of ACM Symposium on Principles of Database Systems*, pages 218–229, April 1990.

[AV87] S. Abiteboul and V. Vianu. A Transaction Language Complete for Database Update and Specification. *Proceedings of ACM Symposium on Principles of Database Systems*, pages 260–268, March 1987.

[AV88] S. Abiteboul and V. Vianu. Procedural and Declarative Database Update Language. *Proceedings of ACM Symposium on Principles of Database Systems*, pages 240–250, March 1988.

[CH80] A.K. Chandra and D. Harel. Computable Queries for Relational Databases. *Journal of Computer and System Sciences*, 21(2):156–178, October 1980.

[Cha81] A.K. Chandra. Programming Primitives for Database Languages. *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 50–62, 1981.

[End72] H.B. Enderton. *A Methematical Introduction to Logic.* Academic Press, 1972.

[GL88] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. *Proceedings of the Fifth Logic Programming Symposium*, pages 1070–1080, 1988.

[GS85] Y. Gurevich and S. Shelah. Fixed-Point Extensions of First-Order Logic. In *Proceedings of 26th Annual Symposium on Foundations of Computer Science*, pages 346–353. IEEE, 1985.

[HS89] R. Hull and J. Su. Untyped Sets, Invention, and Computable Queries. *Proceedings of ACM Symposium on Principles of Database Systems*, pages 347–359, March 1989.

[HS90] R. Hull and J. Su. Algebraic and Calculus Query Languages for Recursively Typed Complex Objects, 1990. to appear in JCSS.

[KN88] R. Krishnamurthy and S. Naqvi. Nondeterministic Choice in Datalog. Technical Report ACA-ST-029-88, MCC, Austin, TX 78759, March 1988.

[KP88] P.G. Kolaitis and C.H. Papadimitriou. Why Not Negation By Fixpoint? *Proceedings of ACM Symposium on Principles of Database Systems*, pages 231–239, March 1988.

[MS88] C. Maindreville and E. Simon. Modelling Non Deterministic Queries and Updates In Deductive Databases. *Proceedings of International Conference on Very Large Databases*, pages 395–406, 1988.

[NT89] S. Naqvi and S. Tsur. *A Logical Language for Data and Knowledge Bases.* Computer Science Press, 1989.

[Prz88a] T.C. Przymusinski. On the Declarative and Procedural Semantics of Logic Programs. *Journal of Automated Reasoning*, 1988.

[Prz88b] T.C. Przymusinski. On the Declarative Semantics of Deductive Databases and Logic Programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann, Los Altos, CA, 1988.

[RBK88] R. Ramakrishnan, C. Beeri, and R. Krishnamurthy. Optimizing Existential Datalog Queries, 1988. Draft.

[Rei83] R. Reiter. Towards a Logical Reconstruction of Relational Database Theory. In M.L. Brodie, J. Mylopoulos, and J. Schmidt, editors, *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases and Programming Languages.* Springer-Verlag, 1983.

[She90a] Y. Sheng. A Non-deterministic Deductive Database Languages with Tuple-identifications. *PhD dissertation*, September 1990.

[She90b] Y. Sheng. IDLOG: Extending the Expressive Power of Deductive Database Languages. *Proceedings of ACM SIGMOD International Conference on Management of Data*, June 1990.

[SZ90] D. Sacca and C. Zaniolo. Stable Models and Non-Determinism in Logic Programs with Negation. *Proceedings of ACM Symposium on Principles of Database Systems*, 1990.

[Gel88] A. Van Gelder. Negation as Failure Using Tight Derivations for General Logic Programs. In J. Minker, editor, Foundations of Deductive Databases and Logic Programming, pages 149–176. Morgan Kaufmann, Los Altos, CA, 1988.

[VGRS88] A. Van Gelder, K.A. Ross, and J.S. Schlipf. Well-Founded Semantics for General Logic Programs. Technical Report UCSC-CRL-88-16, University of California, Santa Cruz, University of California, Santa Cruz, CA 95064, March 1988.

[Zan86] C. Zaniolo. Safety and Compilation of Non-Recursive Horn Clauses. *Proceedings of of the 1st Expert Database Conf.*, 1986.